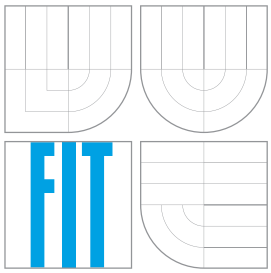
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

AUTOMATIZACE VERIFIKACE ŘÍZENÉ POKRYTÍM PRO PROCESORY ASIP

ASIPS INTELLIGENT TESTBENCH AUTOMATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR
FILIP BADÁŇ

VEDOUCÍ PRÁCE
SUPERVISOR
Ing. MARCELA ZACHARIÁŠOVÁ, Ph.D.

BRNO 2016

Zadání bakalářské práce/18663/2015/xbadan00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2015/2016

Zadání bakalářské práce

Řešitel: **Badáň Filip**
Obor: Informační technologie
Téma: **Automatizace verifikace řízené pokrytím pro procesory ASIP**
ASIPs Intelligent Testbench Automation
Kategorie: Vestavěné systémy

Pokyny:

1. Seznamte se s problematikou funkční verifikace řízené pokrytím v jazyce SystemVerilog podle metodiky UVM.
2. Seznamte se s algoritmem pro automatizaci verifikace řízené pokrytím pomocí genetického algoritmu.
3. Vytvořte návrh, jak aplikovat tento algoritmus pro verifikaci procesorů typu ASIP (Application Specific Instruction-Set Processor) společnosti Codasip.
4. Implementujte tento návrh a experimentálně ověřte na více procesorech.
5. Zhodnoťte dosažené výsledky.

Literatura:

- C. Spear. SystemVerilog for Verification: A guide to Learning the Testbench Language Features, pages 429, Springer 2010, ISBN-13: 978-1441945617.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Šimková Marcela, Ing.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 18. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

doc. Ing. Zdeněk Kotásek, CSc.
vedoucí ústavu

Abstrakt

Táto práca sa zaoberá návrhom a implementáciou automatizácie verifikácie riadenej pokrytím pomocou genetického algoritmu pre aplikačne špecifické procesory. Cieľom práce je prepojiť verifikačné prostredie podľa metodiky UVM s už navrhnutým genetickým algoritmom a pripraviť ho na integráciu do vývojového prostredia Codasip Studio. Jadro finálneho riešenia spočíva v úprave UVM komponentov verifikačného prostredia a v zabezpečení správnej komunikácie genetického algoritmu s generátorom náhodných aplikácií.

Abstract

This thesis focuses on the proposal and implementation of intelligent testbench automation for application-specific processors. The main goal of the thesis is to connect UVM verification environment with already designed genetic algorithm and to prepare this verification environment for integration into Codasip Studio development environment. The core of the final solution is modification of UVM components in verification environment and communication between the genetic algorithm and the generator of random test applications.

Kľúčové slová

funkčná verifikácia, genetický algoritmus, UVM, SystemVerilog, Codasip Studio, ASIP procesory

Keywords

functional verification, genetic algorithm, UVM, SystemVerilog, Codasip Studio, ASIP processors

Citácia

BADÁŇ, Filip. *Automatizace verifikace řízené pokrytím pro procesory ASIP*. Brno, 2016. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Zachariášová Marcela.

Automatizace verifikace řízené pokrytím pro procesory ASIP

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pani Ing. Marcely Zachariášovej, Ph.D. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Filip Badáň
18. mája 2016

Poďakovanie

Týmto by som chcel poďakovať pani Ing. Marcele Zachariášovej, Ph.D. za trpezlivosť a cenné rady pri vypracovávaní práce. Ďalej by som chcel poďakovať spoločnosti Codasip, ktorá prácu zadávala a poskytla pomoc a nástroje na jej vypracovanie.

Obsah

Kapitola 1

Úvod

Verifikácia sa v súčasnej dobe stáva čoraz dôležitejším aspektom návrhu integrovaných obvodov. Každá chyba, ktorá sa dostane až do finálneho produktu, znamená pri obrovskej konkurencií na tomto trhu obrovské finančné straty. Príkladom môže byť nechvalne známa chyba pri delení v procesoroch Pentium z roku 1994, ktorá spoločnosť Intel stála v danom roku až 475 miliónov dolárov [13]. Z tohoto dôvodu sa, podľa najrozsiahlejšej štúdie tohoto priemyslu z roku 2014 [6], verifikácii prikladá veľký význam a sú jej venované nemalé časové aj finančné prostriedky. Z údajov získaných touto štúdiou, zaberá verifikácia v praxi priemerne až 57% z celkového projektového času, pričom počet projektov, kde verifikácia trvala až 80% celkového projektového času sa zvýšil od roku 2012 o 5% na celkových 10%. Integrované elektronické obvody sa v súčasnosti stávajú čoraz zložitejšie a medzi najzložitejšie typy takýchto obvodov patria procesory. Počet procesorov na čipoch a takisto ich komplexnosť neustále rastie a verifikácia sa tým pádom stáva čoraz ťažšia a nákladnejšia.

Najrozšírenejším typom verifikačného prístupu je v súčasnosti funkčná verifikácia, ktorej je venovaná kapitola 3. Pre veľmi jednoduché obvody je možné overiť ich celú funkcionalitu priamymi testami napísanými verifikačnými inžiniermi podľa danej špecifikácie a ručnou kontrolou výstupov. S narastajúcou komplexnosťou návrhov sa však tento spôsob stal priveľmi nákladný a neefektívny. Riešením tohto problému je používanie pseudo-náhodných testov, ktoré dosiahnu požadované pokrytie (t.j. overenie stanovených cieľov verifikácie) oveľa rýchlejšie a verifikačný tím nemusí tvoriť všetky testovacie prípady manuálne. Nevýhodou tohto riešenia je však vysoká redundancia testovacích prípadov a takisto malá šanca na dosiahnutie niektorých špecifických situácií. Preto sa v dnešnej dobe používa najmä kombinácia priamych a pseudo-náhodných testov a verifikácia riadená pokrytím [6]. Verifikácia riadená pokrytím spočíva v analýze dosiahnutého pokrytia po každom behu simulácie a následným upravením vstupov pre dosiahnutie nepokrytých situácií. Takáto úprava môže byť manuálna alebo automatická. Pokrývanie ťažko dosiahnuteľných a špecifických prípadov ručne je však stále náročnejšie a stojí čoraz viac času a prostriedkov. Jedným z riešení je použitie formálnej alebo semi-formálnej verifikácie, pomocou ktorej je možné, vďaka použitiu matematických prístupov na preskúmanie celého stavového priestoru, dosiahnuť stopercentné pokrytie. Pre veľmi zložité a komplexné obvody, ako sú napríklad procesory, je tento prístup však v súčasnosti stále príliš náročný [15].

Efektívnejším a rýchlejším riešením tohto problému je nasadenie inteligentného algoritmu, ktorý na základe zmeraného pokrytia automaticky generuje ďalšie testovacie prípady. Týmto spôsobom je možné docieliť požadované pokrytie výrazne rýchlejšie, bez nutnosti tvoriť testy manuálne a zároveň odstrániť redundanciu spôsobenú pseudo-náhodnými testami. Jediným problémom pri použití automatickej verifikácie riadenej pokrytím je náročná

implementácia riadiaceho algoritmu a jeho integrácia do súčasných, už existujúcich verifikačných prostredí a metodík. Z tohto dôvodu je dôležitá takisto aj automatizácia inteligentne riadených verifikačných prostredí, pre dosiahnutie čo najvyššej znovupoužiteľnosti a tiež ich jednoduchšiu a rýchlejšiu aplikáciu v praxi. Práve toto je kľúčové pre rozšírenie automatizovanej verifikácie riadenej pokrytím v praxi a tým pádom, dôležité pre to, aby boli verifikačné tímy schopné držať tempo s rýchlo rastúcou zložitosťou a komplexnosťou verifikovaných obvodov.

Cieľom tejto práce je integrácia už navrhnutého genetického algoritmu [16], slúžiaceho na automatické generovanie testov pre verifikovaný obvod, do vývojového prostredia Codasip Studio. Práca sa zameriava na aplikáciu tohto algoritmu pre aplikačne špecifické inštrukčné procesory (ASIP, angl. *application-specific instruction-set processor*). Prostredie Codasip Studio, vyvinuté spoločnosťou Codasip slúži mimo iného, aj na automatické generovanie verifikačného prostredia pre ASIP procesory v ňom navrhnuté. Toto verifikačné prostredie je navrhnuté podľa metodiky UVM (angl. *Universal Verification Methodology*, podrobnejšie v kapitole 2) vyvinutou pod záštitou organizácie Accellera, ktorá umožňuje efektívnu implementáciu a vysokú znovupoužiteľnosť verifikačného prostredia. Genetický algoritmus je implementovaný ako nadstavba pre toto generované verifikačné prostredie a na jeho úspešnú integráciu bolo potrebné navrhnúť a implementovať nasledujúce kroky, ktoré tvoria hlavnú náplň tejto práce:

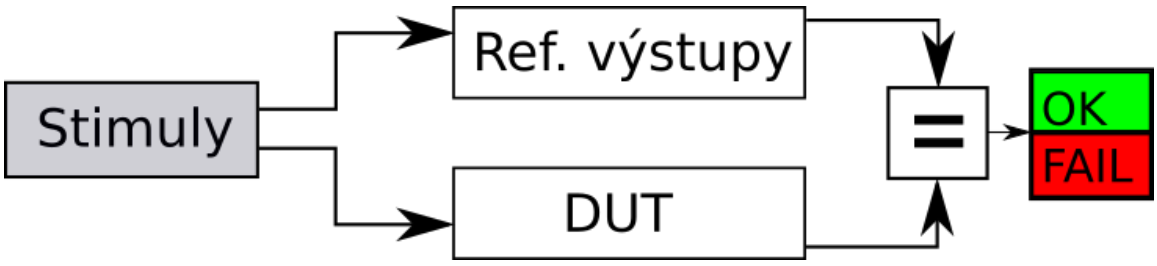
- Úprava komponent UVM v ktorých je implementované jadro GA pre účely verifikácie.
- Sada Tcl skriptov určených na riadenie behu verifikácie a vyhotovenie výslednej správy o výsledkoch verifikácie.
- Integrácia generátor náhodných aplikácií od spoločnosti Codasip do verifikačného prostredia a jeho prepojenie s GA.

V kapitole 2 sú popísané základy funkčnej verifikácie a metodiky UVM. História a princíp genetických algoritmov je vysvetlená v kapitole 3. Kapitola 4 popisuje návrh automatizácie verifikačného prostredia riadenej pokrytím a integrácie genetického algoritmu. V kapitole 5 je objasnená implementácia a realizácia celého riešenia a v kapitole 6 sú zhodnotené experimenty a zobrazené dosiahnuté výsledky.

Kapitola 2

Funkčná verifikácia

Hlavným cieľom funkčnej verifikácie je ukázať, že verifikovaný obvod spĺňa požadovanú špecifikáciu. Na základe tejto špecifikácie HW návrhár implementuje daný obvod a verifikačný inžinier pripraví vstupy, respektíve stimuly pre verifikáciu a k nim referenčné výsledky alebo referenčný model. Princíp funkčnej verifikácie potom spočíva v porovnaní výstupov verifikovaného obvodu s referenčnými výstupmi, prípadne výstupmi referenčného modela pre rovnaké stimuly. Tento princíp je znázornený na obrázku 2.1. Hlavným dôvodom prečo je dôležité aby verifikační inžinieri a hardvéroví návrhári boli rôzny ľudia, je rozdielne pochopenie špecifikácie. Špecifikácia je väčšinou slovná a preto je možné, že obsahuje rôzne nepresnosti alebo nejasnosti a z toho dôvodu hrozí riziko rozdielneho pochopenia niektorých jej častí. Ak by návrhár navrhol obvod na základe zle pochopenej špecifikácie, pri následnej verifikácií tohto obvodu vykonanej týmto istým človekom, by takto zapríčinené chyby neodhalil. Ďalším problémom, ktorý musia verifikačné tímy riešiť je veľká zložitosť a široká funkcionalita verifikovaných obvodov. Verifikácia zaberá väčšinu času a prostriedkov z celkového návrhu integrovaného obvodu, o čom svedčí aj veľký nárast ľudí pracujúcich v tejto oblasti, kde počet verifikačných inžinierov už v priemere presiahol počet návrhárov [6].



Obr. 2.1: Grafické znázornenie procesu funkčnej verifikácie

Pojem funkčná verifikácia v sebe zahŕňa komplexný proces overovania, či sa daný obvod správa tak ako je od neho očakávané. Je založená na simulácií činnosti obvodu pomocou simulačných nástrojov (ako napríklad QuestaSim od spoločnosti Mentor Graphics) a jej následným vyhodnotením. Jedným z najdôležitejších krokov verifikácie je určenie, kedy môžeme verifikáciu považovať za ukončenú a úspešnú. Funkčnou verifikáciou, na rozdiel od formálnej verifikácie, nikdy nemôžeme stopercentne dokázať, že je verifikovaný obchod bez chyby. Naopak, dokázať môžeme len prítomnosť nejakej chyby v obvode. Preto je pre verifikačný tím dôležité určiť ciele verifikácie. Na tento účel slúžia metriky pokrytia, po-

mocou ktorých je možné určiť, akým spôsobom chceme overiť funkčnosť obvodu. Dvoma základnými typmi pokrytia sú podľa [15]:

- **Štrukturálne pokrytie**, alebo aj pokrytie kódu (angl. *structural/code coverage*) zahŕňajúce pokrytie príkazov (angl. *statement coverage*), pokrytie výrazov (angl. *expression coverage*), pokrytie ciest (angl. *path/branch coverage*), pokrytie podmienok (angl. *condition coverage*) a pokrytie stavov a prechodov konečného automatu (angl. *FSM coverage*)
- **Funkčné pokrytie** (angl. *functional coverage*), ktoré zahŕňa sémantické časti funkcionality. Toto pokrytie je obvykle tvorené verifikačným tímom podľa zadanej špecifikácie obvodu.

Ďalším krokom je implementácia verifikačného prostredia pre daný obvod. Tvorba nového verifikačného prostredia pre každý verifikovaný obvod by však bola časovo veľmi náročná a preto vznikajú rôzne metodiky ako napríklad UVM (Universal Verification Methodology) alebo OVM (Open Verification Methodology). Metodike UVM sa budeme bližšie venovať neskôr v podkapitole 2.2.

Po implementácií verifikačného prostredia nastáva samotné overovanie funkčnosti dizajnu. Toto sa väčšinou vykonáva pomocou simulačných nástrojov, ktoré dokážu spustiť simuláciu a spracovať dosiahnuté pokrytie. Medzi takéto nástroje patrí napríklad QuestaSim od spoločnosti Mentor Graphics. Generovanie vstupov je zabezpečené pseudo-náhodnými testami riadenými nejakou obmedzujúcou logikou (angl. *constrained random tests*), prípadne priamymi testami (angl. *directed tests*). Po skončení simulačného behu môže nastať vyhodnotenie dosiahnutého pokrytia a to buď manuálne alebo automaticky. Na základe získaných informácií sú upravené podmienky pre náhodne generované testy alebo dopísané, či modifikované priame testy. Takýto spôsob verifikácie, ktorý je v súčasnosti najpoužívanejší sa nazýva **verifikácia riadená pokrytím** (angl. *coverage-driven verification*).

Na overenie výstupov verifikovaného obvodu sa používajú podľa [15] tri základné prístupy:

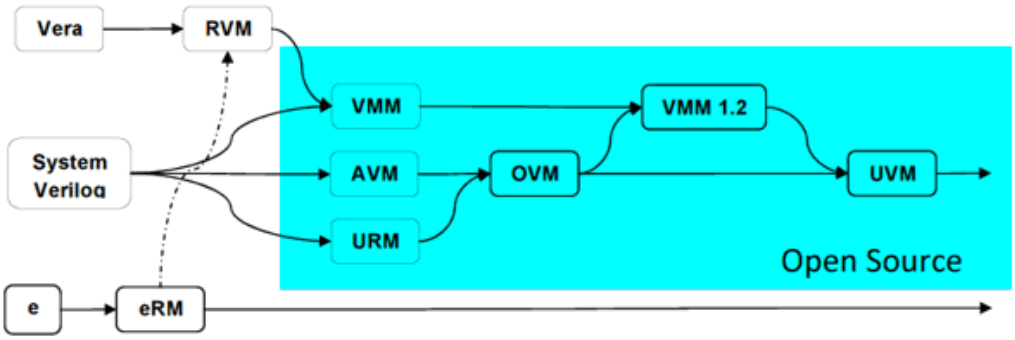
- **Referenčný vektor** (angl. *golden vector*), ktorý obsahuje väčšinou manuálne nastavené referenčné výstupy pre dané vstupné stimuly. Tento vektor sa potom porovnáva s výstupmi verifikovaného obvodu.
- **Referenčný model** (angl. *golden/reference model*) implementovaný ako abstraktný model v nejakom vyššom programovacom jazyku. Tomuto modelu sú poskytnuté vstupy rovnaké ako pre verifikovaný obvod DUT (Design Under Test) a následne porovnávané výstupy. Aby mohli byť tieto výstupy kontrolované počas každého cyklu, musí byť model implementovaný ako "cycle-accurate", čo znamená, že musí produkovať výstupy súbežne s verifikovaným obvodom.
- **Overovanie založené na transakciách** (angl. *transaction-based checking*), kde sú očakávané výstupy uložené ako transakcie do tzv. scoreboardu a následne porovnávané s výstupmi verifikovaného obvodu.

V praxi sa používa aj kombinácia týchto prístupov.

2.1 Programovacie jazyky a metodiky

Na implementáciu verifikačných prostredí sa v dnešnej dobe používajú prevažne jazyky Verilog, SystemVerilog, Specman e, VHDL a Vera. Najmä jazyk SystemVerilog získal v po-

sledných rokoch veľkú popularitu a jeho použitie vo verifikačnom priemysle sa značne zvýšilo [6]. SystemVerilog je programovací jazyk kombinujúci jazyk na popis hardvéru (HDL - Hardware Description Language) a jazyk slúžiaci na verifikáciu. Bol vyvinutý organizáciou Accellera v roku 2002 ako rozšírenie jazyka Verilog. Dôvodom vzniku tohto jazyka bolo zjednodušenie návrhu a hlavne verifikácie komplexných obvodov. Jedným z cieľov Accellery bolo aj zachovanie spätnej kompatibility a čo najvyššej podobnosti s jazykom Verilog. Medzi kľúčové body, ktoré rozširujú funkcionalitu HDL jazykov o verifikačné vlastnosti patrí najmä možnosť definície obmedzujúcich podmienok (angl. *constraints*) pre generovanie náhodných stimulov a takisto podpora tvorby bodov funkčného pokrytia (angl. *coverpoints*). Práve tieto vlastnosti sa mimo iného zaslúžili o obrovskú popularitu tohto jazyka vo verifikačnej komunite [14]. Potreba zaviesť tradičné programovacie princípy ako napríklad OOP a zjednodušiť návrh a implementáciu viedla ku vzniku verifikačných metodík. Prvými významnými verifikačnými metodikami boli proprietárne eRV (*e* Reuse Methodology) pre jazyk *e* a RVM (Reuse Verification Methodology) pre jazyk Vera. Tieto metodiky poskytli základ všetkým ďalším metodikám, pretože predstavili základné komponenty verifikačného prostredia používané dodnes, ako sú scoreboardy, monitory alebo sekvencie. S rastúcou popularitou jazyka SystemVerilog sa metodika RVM preorientovala na tento jazyk z čoho vznikla VVM (Verification Methodology Manual) od spoločnosti Synopsys. Následne v roku 2006 spoločnosť Mentor Graphics predstavila vlastnú metodiku postavenú na jazyku SystemVerilog nazvanú AVM (Advanced Verification Methodology). AVM bola dôležitá hlavne z toho dôvodu, že bola ako prvá poskytnutá ako open-source riešenie. Spoločnosť Cadence vyvinula v roku 2007 takisto SystemVerilogovú open-source verziu eRV nazvanú URM (Universal Reuse Methodology). V januári 2008 spojili Cadence a Mentor Graphics sily a spoločne vyvinuli novú jednotnú open-source metodiku OVM (Open Verification Methodology) pre jazyky *e*, SystemVerilog a System C. OVM získala veľkú popularitu pretože zjednocovala to najlepšie z predchádzajúcich metodík od Cadence a Mentor Graphics. Táto metodika sa spolu s pridaním princípov metodiky VVM od Synopsysu stala základom zatiaľ najmodernejšej metodiky zvanej UVM (Universal Verification Methodology) vyvinutej pod organizáciou Accellera [7] [1]. Vývoj verifikačných metodík je ešte raz zachytený na obrázku 2.2 [2].

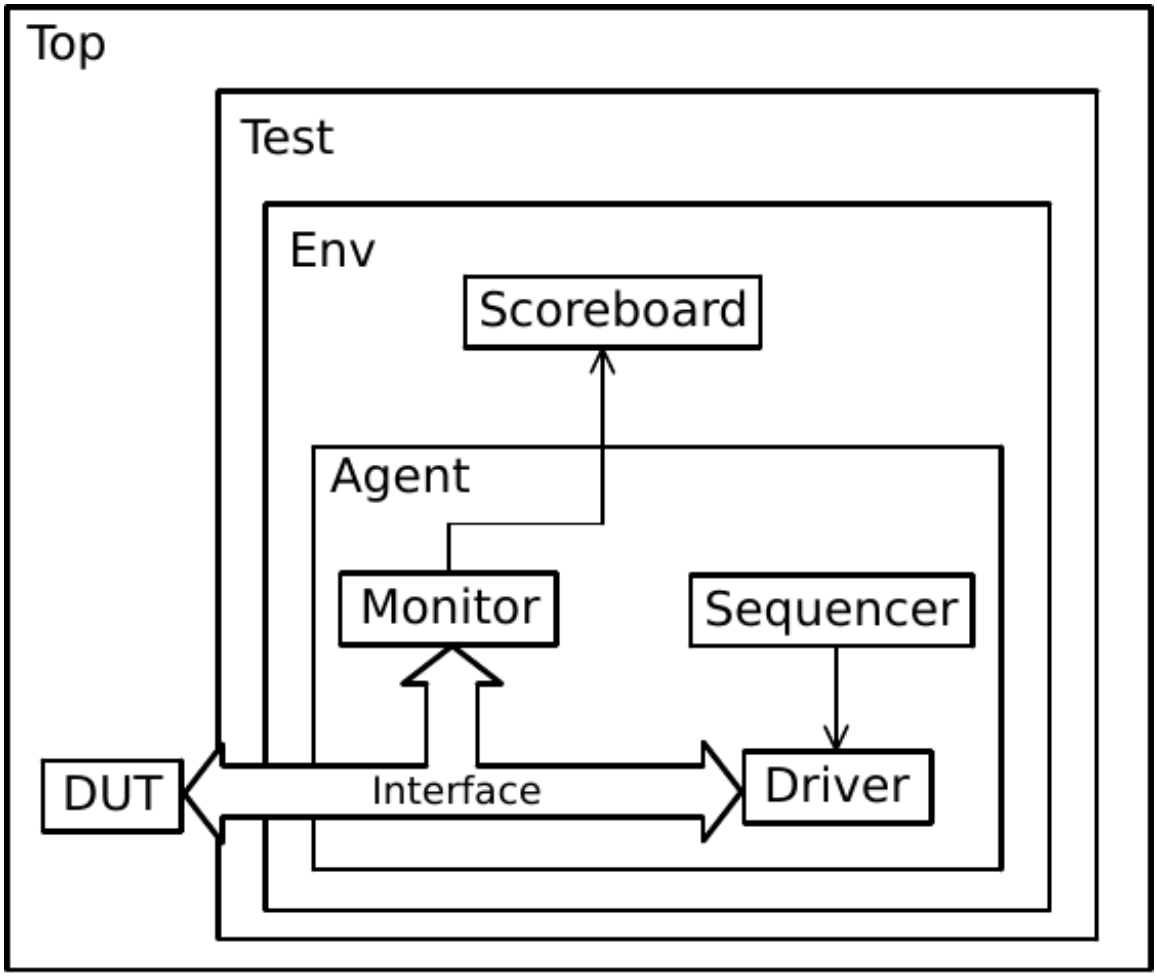


Obr. 2.2: História vývoja verifikačných metodík

2.2 UVM

UVM (Universal Verification Methodology) umožňuje tvorbu efektívnych verifikačných prostredí a vysokú znovu použiteľnosť jednotlivých komponentov prostredia. Je poskytovaná ako open-source knižnica priamo od Accellery. Je založená na jazyku SystemVerilog a je

podporovaná všetkými významnými simulátormi, ktoré podporujú aj SystemVerilog. Toto je jeden z dôvodov, prečo sa z nej stala v súčasnosti najpoužívanejšia metodika vo verifikačnej komunite [6]. Verifikačné prostredie podľa UVM sa skladá zo základných komponentov nazývaných aj UVC (Universal Verification Components), ktoré sú pripojené cez definované virtuálne rozhranie k verifikovanému obvodu DUT. Komponenty UVM sú navrhnuté tak, aby boli jednoducho modifikovateľné bez potreby meniť celé verifikačné prostredie a aby boli pripravené na ľahké a rýchle znovupoužitie. Na obrázku 2.3 [1] sú zobrazené základné komponenty UVM a ich zapojenie. V ďalšom texte sú jednotlivé komponenty stručne popísané.



Obr. 2.3: Verifikačné prostredie UVM

Transakcia (angl. *transaction/data item*)

Transakcia je základná jednotka prostredia, ktorá zaobaľuje vstupné dáta pre DUT. Trieda transakcie môže obsahovať hodnoty, obmedzujúce podmienky alebo aj metódy vykonávajúce operácie s týmito dátami ako napríklad metóda *print* na vypísanie obsahu transakcie alebo metóda *compare* na porovnanie obsahu transakcií. Obsah transakcie je odvodený zo špecifikácie DUT, zvyčajne je to model komunikácie. Príkladom môže byť objekt, ktorý modeluje komunikačnú zbernicu obsahujúcu dáta a adresu, sieťový paket alebo inštrukciu pre procesor.

Sequencer a sekvencia

Sekvencia je usporiadaná kolekcia transakcií, ktoré budú zaslané ako vstupy verifikovanému obvodu DUT. Každej sekvencií je možné definovať postup, podľa ktorého sa budú jednotlivé transakcie napĺňať. Sequencer je komponent, ktorý preposiela vygenerované transakcie postupne driveru, podľa definovaných sekvencií.

Driver

Hlavnou funkciou driveru je získať transakciu zo sequencera a poslať dáta DUT. Driver rozdelí transakciu a transformuje dáta na signály DUT. Keďže sledovanie tejto interakcie verifikačného prostredia a DUT je zabezpečené komponentom zvaným monitor, funkcionalita driveru by preto mala byt minimalizovaná na odosielanie vstupov pre DUT.

Monitor

Úlohou monitora je sledovať komunikáciu medzi verifikačným prostredím a verifikovaným obvodom. Monitor je pasívna súčasť UVM prostredia, čo znamená, že neposiela žiadne vstupy, ani negeneruje žiadne výstupy. Monitor by mal sledovať vstupy a výstupy zasielané DUT, ktoré ďalej preposiela do komponentu scoreboard. Verifikačné prostredie môže obsahovať aj niekoľko nezávislých monitorov.

Scoreboard

Scoreboard je veľmi dôležitou súčasťou verifikačného prostredia. Jeho úlohou je získavať transakcie DUT z monitora a následne ich porovnávať s referenčnými hodnotami. Scoreboard potom následne podáva hlásenie o výsledku tohto porovnania.

Agent

Hlavnou príčinou vzniku komponentu agent je ľahšie znovupoužitie monitoru, sequenceru a driveru. Agent prepája a zapúzdruje tieto komponenty, čím znižuje prácu potrebnú na ich opätovné použitie a zvyšuje prehľadnosť výsledného kódu. Verifikačné prostredie môže obsahovať viacero agentov, typicky sa používa jeden agent na jedno logické rozhranie DUT. Agenti sa delia na pasívnych a aktívnych, kde aktívni agenti riadia vstupy a výstupy DUT a pasívni len monitorujú komunikáciu.

Prostredie

Prostredie (angl. *environment*) je základný komponent UVM. Obsahuje jedného alebo viacerých agentov a môže obsahovať aj monitor, nevzťahujúci sa k aktivite žiadneho konkrétneho agenta. Tento komponent by mal byť kvôli prehľadnosti a stále zdôrazňovanej znovupoužiteľnosti, konfigurovateľný pomocou nastaviteľných vlastností.

Test a Top-level

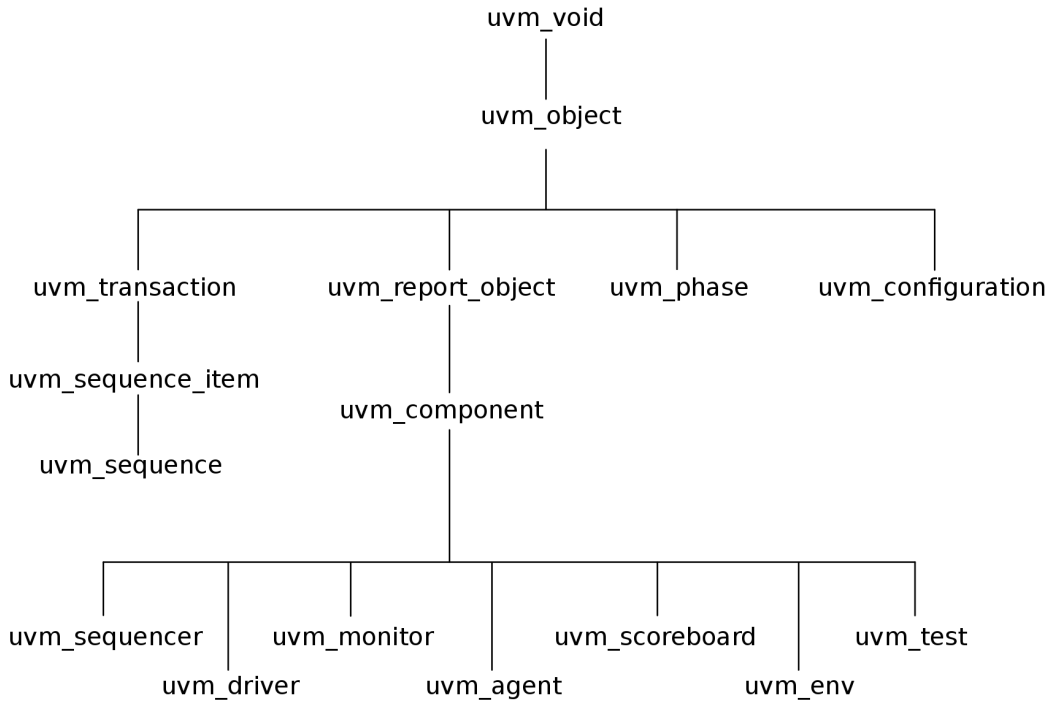
V teste sa vytvára inštancia prostredia a DUT a spúšťa sa testovacia fáza. Top-level zabezpečuje prepojenie verifikačného prostredia s rozhraním verifikovaného obvodu DUT.

Triedy a makrá UVM

Metodika UVM využíva základné princípy OOP programovania a jeho výhody. Obrázok č. 2.4 [1] zobrazuje stromovú hierarchiu najdôležitejších UVM tried. Všetky UVM komponenty sú odvodené z už preddefinovaných UVM tried. Toto umožňuje veľmi efektívne prepojenie komponentov a takisto jednoduchú komunikáciu medzi nimi. Triedy takisto ob-

sahujú užitočné (už predimplementované) metódy na prácu s nimi. Verifikačný inžinier sa tak môže sústrediť na zmeny v tých častiach verifikačného prostredia, ktoré sa týkajú konkrétnej funkcionality DUT. Ďalším dôležitým aspektom v UVM sú makrá. Makrá UVM definujú niektoré užitočné metódy tried a premenných. Používanie makier nie je nevyhnuté, avšak doporučuje sa. Medzi najdôležitejšie makrá patria:

- uvm_component_utils - toto makro zaregistruje novú triedu. Používa sa v objektoch odvodených od uvm_component.
- uvm_objecy_utils - má rovnakú funkciu jako uvm_component_utils, ale používa sa v objektoch odvodených od uvm_object.
- uvm_info - slúži na výpis správ počas behu simulácie.
- uvm_error - toto makro posiela chybové výpisy na výstup simulácie.

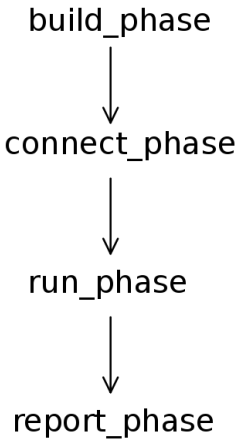


Obr. 2.4: Štruktúra najdôležitejších UVM tried

UVM fázy Priebeh testu v UVM je rozdelený na tzv. fázy. Fázy sú implementované ako metódy v každej UVM triede. Tieto metódy sa vykonávajú postupne pre každú triedu, podľa toho v akej fáze sa momentálne nachádza verifikačné prostredie. Základné fázy a ich priebeh sú zobrazené na obrázku č. 2.5 a popísané v nasledujúcom texte [1].

- Fáza build (angl. *build phase*) sa používa na vytvorenie inštancií jednotlivých tried. Build fáza agenta teda napríklad zostrojí komponenty monitor, driver a sequencer.
- Fáza connect (angl. *connect phase*) slúži na prepojenie jednotlivých komponentov prostredia. Connect fáza agenta by teda napríklad prepojila sequencer a driver a pripojila monitor na externý port.

- Fáza run (angl. *run phase*) je hlavná fáza. Počas tejto fázy sa vykonáva celá simulácia.
- Fáza report (angl. *report phase*) sa vykonáva po skončení simulácie a zobrazuje výsledky a výstupy testov.



Obr. 2.5: Základné fázy UVM

Existuje aj mnoho ďalších fáz, nie všetky sú však povinné alebo vždy dôležité. Fázy, ktoré sa v daných triedach nepoužívajú, v nich nemusia byť definované.

Kapitola 3

Genetické algoritmy

Genetické algoritmy (GA) patria medzi najrozšírenejší typ evolučných algoritmov. Hlavná myšlienka evolučných algoritmov je aplikácia Darwinovej teórie o vývoji druhov na riešenie optimalizačných alebo simulačných úloh, pri ktorých by bolo prehľadávanie celého stavového priestoru možných riešení priveľmi zdĺhavé a neefektívne.

Táto myšlienka sa začala objavovať už v 40. rokoch 20. storočia, no bez výkonnej výpočtovej techniky nebolo možné simulovať tisíce evolučných cyklov, ktoré prebiehajú v prírode. Počas 60-tych rokov sa začali na rôznych miestach vyvíjať rozličné prístupy k tejto myšlienke. V USA bolo predstavené evolučné programovanie (angl. *evolutionary computing*) trojicou Fogel, Owens a Walsh [5], zatiaľ čo Holland priniesol svoju verziu nazvanú genetický algoritmus (angl. *genetic algorithm*) [8]. V Nemecku, Rachenberg a Schwefel [11, 12], vyvinuli evolučné stratégie (angl. *evolution strategies*). Ďalšou verziou bolo potom až v roku 1990 genetické programovanie (angl. *genetic programming*) [9], predstavené Kozom. Algoritmy vyvinuté jednotlivými prístupmi boli nakoniec zjednotené pod pojmom evolučné algoritmy [10].

Evolúcia v prírode podľa Darwinovej teórie prebieha spôsobom boja o prostriedky potrebné na život v rámci celej populácie. Tento boj obvykle vyhrajú práve najschopnejší a najsilnejší jedinci, čo im umožní ďalej sa páriť a tým šíriť svoju genetickú informáciu medzi svojich potomkov. Vďaka tomuto prirodzenému výberu, sa populácia stáva stále lepšie pripravená na prežitie. Toto je základný princíp evolučnej teórie nazývaný aj "prežitie najlepších"(angl. *survival of the fittest*) [3]. Pri zasadení tohto princípu do kontextu riešenia problémov, populácia predstavuje množinu možných tzv. kandidátnych riešení daného problému a to ako dobre vedia problém vyriešiť, určuje ich kvalitu.

Hlavná výhoda evolučných algoritmov spočíva práve v jednoduchosti, ale zároveň aj účinnosti tohto princípu.

3.1 Princíp genetických algoritmov

Základnými procesmi genetických algoritmov, ale aj evolučných algoritmov všeobecne, je najmä obmena jedincov pomocou použitia dvoch genetických operátorov a to kríženie (angl. *crossover/recombination*) a mutácia (angl. *mutation*) a tiež selekcia jedincov, ktorí sa dostanú do ďalšej generácie. Keďže evolúcia je stochastický proces je treba zdôrazniť, že veľkú rolu v evolučných algoritmoch hrá aj istá náhodnosť týchto procesov. Práve kvôli tomuto nie je jednoduché dokázať konvergenciu evolučných algoritmov, no aplikácia v praxi ukazuje väčšinou veľmi pozitívne výsledky [4]. Táto náhodnosť je však dôležitá z dôvodu,

aby aj menej kvalitný jedinci mali šancu stať sa rodičmi, alebo prežiť do ďalšej generácie. Vďaka tomu GA nesmeruje len k lokálne najlepšiemu riešeniu, ale väčšinou postupne konverguje ku globálnemu optimu.

Aby bolo možné aplikovať evolučnú teóriu do výpočtovej oblasti, je nutné reprezentovať prvky reálneho sveta do prostredia riešeného problému. Z tohto pohľadu je dôležitá najmä reprezentácia jedincov a toto je miesto, kde sa jednotlivé typy evolučných algoritmov najvýraznejšie odlišujú. Pri genetických algoritmoch sú vlastnosti riešenia vyjadrené pomocou génov. Tieto gény sa spájajú a tým vytvoria reťazec nazývaný chromozóm. Chromozómy zastupujú (kódujú) kandidátnych jedincov, ktorí predstavujú potencionálne riešenie s určitými vlastnosťami, inak povedané, rôzne hodnoty alebo parametre daného riešenia. GA obvykle využívajú binárnu reprezentáciu, čo znamená, že jednotlivé chromozómy sú zakódované ako bitový reťazec. Základným algoritmom GA je [4]:

Algorithm 1 Základný genetický algoritmus

```
1: procedure GA
2:     inicializácia GA
3:     vyhodnotenie každého jedinca
4:     while splnená podmienka ukončenia do
5:         výber rodičov
6:         kríženie
7:         vyhodnotenie nových jedincov
8:     end while
9: end procedure
```

Kvalita (angl. *fitness*) daného jedinca sa vyhodnocuje pomocou tzv. *fitness funkcie*. Táto funkcia úzko súvisí s konkrétnym riešeným problémom a jej úlohou je ohodnotiť vstupný chromozóm predstavujúci jedno riešenie podľa toho ako kvalitne rieši daný problém.

Selekcia rodičov je sčasti náhodná, ale rolu tu hrá aj kvalita jedinca. Pridelenie pravdepodobnosti s akou sa jedinec stane rodičom ďalšej generácie môže závisieť od jeho kvality v porovnaní s kvalitou zvyšku populácie. Na základe týchto pravdepodobností sa vyberú noví rodičia pomocou rulety tzv. *roulette wheel* algoritmom. Princíp tohto algoritmu spočíva v rozdelení rulety pre jednotlivé chromozómy podľa ich kvality, takže kvalitnejšie chromozómy majú väčšiu šancu na úspech. Druhým spôsobom je tzv. *turnajový výber* (angl. *tournament selection*), kde sa náhodne vyberie istý počet jedincov a z nich najkvalitnejší je určený na kríženie. Počet vybratých jedincov je dopredu stanovený a čím je vyšší, tým menšiu šancu majú menej kvalitné chromozómy.

Kríženie alebo aj re-kombinácia rodičov je dôležitou súčasťou genetických algoritmov. Jeho myšlienka spočíva vo výmene niektorých génov obvykle dvoch vybratých rodičov. Po tejto výmene môže vzniknúť potomok, ktorý má rovnakú alebo aj nižšiu kvalitu ako jeho rodičia, ale je tiež dobrá šanca, že takto vznikne kombinácia tých najlepších génov z oboch rodičov a tým pádom výrazne kvalitnejší jedinec. V bitovej reprezentácií používanej v GA prebieha kríženie ako rozdelenie chromozómu rodičov na jednej (*one-point*) alebo viacerých (*N-point*) pozíciách a následná náhodná výmena niektorých takto vytvorených segmentov. Týmto spôsobom môže vzniknúť jeden alebo aj viacero potomkov. Ďalším možným prí-

stupom je tzv. *uniformné* kríženie. Pri tomto spôsobe sa každý gén potomka vyhodnocuje osobitne s dopredu určenou pravdepodobnosťou s akou bude gén vybratý z jedného alebo druhého rodiča. Táto pravdepodobnosť býva obvykle 0.5, aby mali výslední potomkovia vyvážený počet génov z oboch rodičov.

Mutácia v bitovej reprezentácií prebieha obvykle ako jednoduchá inverzia niektorých náhodne vybratých bitov. Počet vybratých bitov nie je dopredu určený, ale závisí od dĺžky reťazca. Existujú aj ďalšie možné spôsoby využívajúce napríklad rôzne rozloženia pravdepodobnosti na bity reťazca.

Výber novej generácie spočíva vo vytvorení nových jedincov pomocou mutácie a kríženia dvoch či viacerých rodičov. Novo-vytvorení jedinci potom tvoria novú generáciu a nahradia svojich rodičov. Tento model populácie sa nazýva *generačný* (angl. *generational model*). Ďalšou možnosťou je výber niekoľkých jedincov z predchádzajúcej generácie na základe ich veku (tzn. počet cyklov v ktorých sú súčasťou generácie) alebo ich kvality. Tento model sa nazýva *postupný model* (angl. *steady-state model*).

Ideálna **podmienka ukončenia** činnosti GA je nájdenie globálneho optima, čiže najlepšieho možného riešenia daného problému. Avšak vzhľadom na to, že GA využíva z veľkej časti stochastické procesy, nie je zaručené nájdenie tohto riešenia. Z tohoto dôvodu sa ukončuje buď splnením tejto podmienky, alebo splnením podmienky, ktorá bude po istom čase určite vždy splnená. Ako takéto podmienky sa používajú najmä:

- Dosiahnutie určeného časového limitu.
- Dosiahnutie istého počtu cyklov GA.
- Hodnota najkvalitnejšieho jedinca sa po dlhší čas výrazne nezlepšuje.
- Diverzita populácie dosiahne stanovený limit.

3.2 Parametrizácia genetických algoritmov

Hlavná konfigurácia genetického algoritmu pozostáva z určenia spôsobu reprezentácie dát, spôsobu kríženia, spôsobu mutácie, algoritmu pre výber novej generácie a modelu populácie. Ďalšie parametre potrebné pre kompletnú konfiguráciu sú veľkosť populácie, pravdepodobnosti jednotlivých procesov a prípadne počet jedincov určených na turnajový výber. Vhodná konfigurácia závisí od konkrétneho riešeného problému a môže byť rozhodujúca aj pre nájdenie či nenájdenie dostatočne dobrého riešenia. Typickým prístupom zisťovania správnej konfigurácie je experimentálne testovanie GA s rôznymi parametrami a porovnávanie nameraných výsledkov.

3.3 Vlastnosti GA a zhrnutie

GA sú v súčasnosti najrozšírenejšie evolučné algoritmy a používajú sa v rôznych oblastiach ako napríklad návrh a optimalizácia elektrických obvodov, robotika, učenie neurónových sietí a mnoho ďalších. GA prehľadávajú stavový priestor stochasticky a na viacerých miestach súčasne a práve preto dosahujú vo väčšine prípadov dobré výsledky. Takisto jednoduchosť a efektivita hlavného princípu je ďalším dôvodom ich popularity. Nevýhodou GA

je v prvom rade veľká časová náročnosť. Simulácia často krát až mnoho tisícov evolučných cyklov zaberá veľa času a výpočtových prostriedkov, no s neustále sa zlepšujúcimi technológiami tento problém prestáva byť tak výrazný. Ďalšou nevýhodou je aj fakt, že nie je zaručené nájdenie najlepšieho riešenia, práve kvôli náhodnosti väčšiny procesov.

Kapitola 4

Návrh riešenia

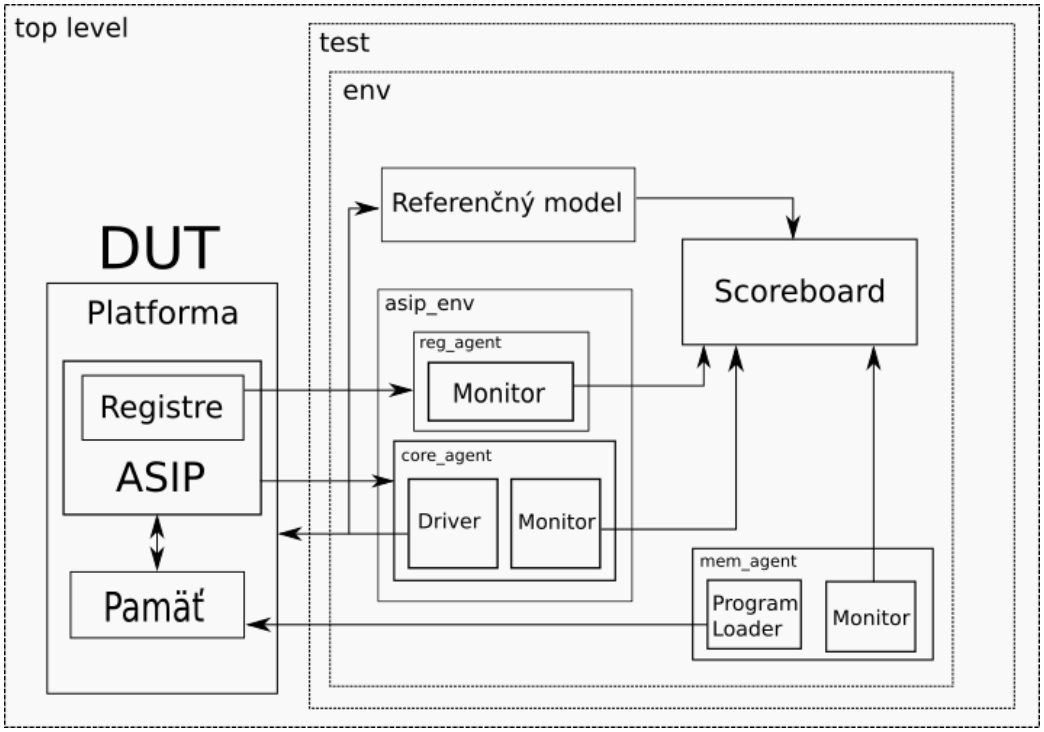
Cieľom tejto práce je navrhnúť a implementovať riešenie pre automatizáciu verifikácie riadenej pokrytím, ktorá je implementovaná pomocou genetického algoritmu. Aplikačnou doménou pre túto automatizáciu a optimalizáciu je verifikácia procesorov typu ASIP vo vývojovom prostredí Codasip Studio od spoločnosti Codasip. Veľkú rolu v riešení tejto úlohy teda zohráva integrácia už existujúceho GA [16] do verifikačného prostredia, automatická úprava komponent GA podľa typu verifikovaného procesoru a konfigurácia parametrov GA z prostredia Codasip Studia.

4.1 Verifikačné prostredie v Codasip Studiu

Codasip Studio (CS) je primárne určené na návrh a prácu s procesormi typu ASIP. Jednou z funkcií Codasip Studia je aj automatické generovanie verifikačného prostredia pre navrhnutý procesor podľa metodiky UVM. Schéma verifikačného prostredia pre základnú platformu obsahujúcu jadro procesora ASIP a pripojenej pamäte je zobrazená na obr. 4.1.

Verifikačné prostredie obsahuje základné UVM komponenty. Existuje tu niekoľko pasívnych agentov, ktorých úlohou je monitorovať činnosť na jednotlivých komponentoch procesora ako sú registre, pamäť a prípadne ďalšie zapojené komponenty. Monitory týchto agentov zasielajú informáciu do komponentu *scoreboard*, kde sa výsledok porovnáva s výstupmi referenčného modelu. Monitory taktiež získavajú informácie o dosiahnutom pokrytí. Súčasťou CS je aj generátor náhodných aplikácií, pomocou ktorého je možné vygenerovať aplikácie pre konkrétny model procesoru. Aplikácia slúžiaca ako test pre verifikovaný procesor je potom nahratá do verifikačného prostredia a v ňom pomocou komponentu *program loader* nahratá do pamäte odkiaľ si ju procesor postupne načítava. Tak isto je daná aplikácia poskytnutá ako vstup aj referenčnému modelu. Ako referenčný model v prostredí CS slúži inštrukčný model daného procesoru v architekturalnom jazyku CodAL, poprípade model procesoru implementovaný užívateľom.

CS ponúka tiež prepojenie verifikačného prostredia so simulátorom QuestaSim od spoločnosti Mentor Graphics. Na tento účel je spolu s verifikačným prostredím vygenerovaná sada Tcl skriptov, slúžiaca na kompiláciu zdrojových kódov, spustenie simulácie pomocou QuestaSim a zobrazenie výsledkov verifikácie, ako aj dosiahnutého pokrytia a priebehu hodnôt jednotlivých signálov v DUT a v referenčnom modeli.

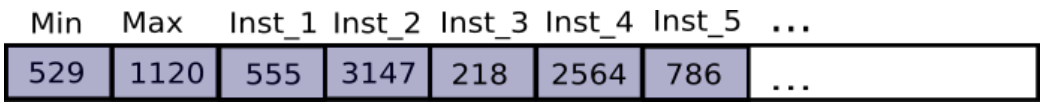


Obr. 4.1: Verifikačné prostredie pre procesorovú platformu

4.2 Genetický algoritmus pre ASIP procesory

Na obrázku 4.3 je znázornené zapojenie genetického algoritmu navrhnutého v [16] do UVM prostredia vygenerovaného pomocou CS. Do procesu verifikácie je taktiež zapojený aj generátor náhodných aplikácií, ktorý je súčasťou CS. Tento generátor vytvára aplikácie zložené z inštrukcií pre procesor navrhnutý v CS. Inštrukčnú sadu a obmedzenia pre jednotlivé inštrukcie vie získať zo súborov obsahujúcich sémantický popis inštrukcií, vygenerovaných pomocou CS z návrhu procesoru. Ako obmedzenia pre generovanie inštrukcií môžeme chápať rôzne inicializácie registrov pred vykonaním inštrukcie, definíciu návestí pre skokové inštrukcie a podobne.

Hlavný princíp zabudovania GA do verifikácie je úprava váh s akou generátor generuje jednotlivé inštrukcie do rôznych aplikácií. Chromozóm GA teda obsahuje váhu pre každú inštrukciu procesoru, s akou bude daná inštrukcia generovať a tiež minimálny a maximálny počet inštrukcií výslednej aplikácie. Na základe jedného chromozómu sa vygeneruje jedna aplikácia pre procesor. Obrázok 4.2 znázorňuje takýto chromozóm s rôznymi váhami pre každú inštrukciu. Kvalita každého chromozómu sa odvíja od výsledného pokrytia dosiahnutého verifikáciou pre aplikáciu vytvorenú na základe tohto chromozómu. Toto vyhodnotenie sa vykonáva verifikačným behom, typicky pomocou RTL simulátora, ako napríklad QuestaSim, pre túto aplikáciu. Výsledné pokrytie je teda vstupom pre genetický algoritmus, ktorý

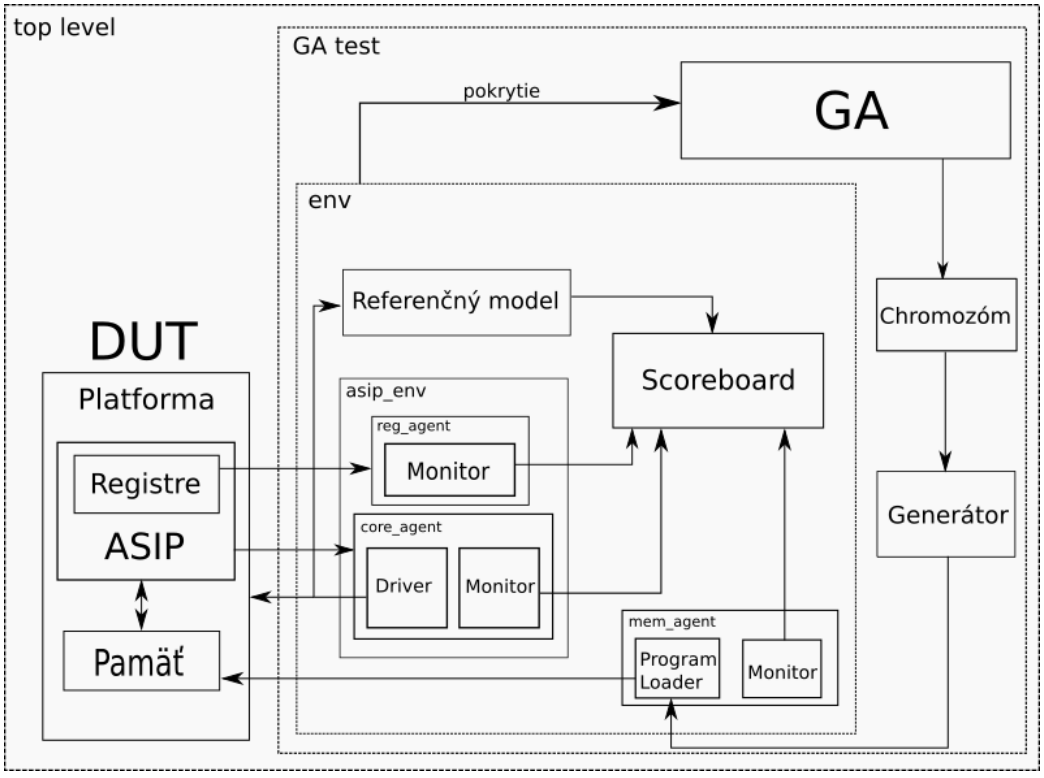


Obr. 4.2: Grafické znázornenie chromozómu

na základe tohto pokrytia zase ovplyvňuje generovanie aplikácií pre verifikáciu, tak ako je vidieť na obrázku 4.3. Takouto spätnou väzbou pre generátor je možné výrazne zoptimalizovať počet vygenerovaných testovacích aplikácií potrebných na dosiahnutie požadovaného pokrytia, pretože GA sa rýchlejšie zameriava na nepokryté oblasti. Pri uchovaní aplikácií vytvorených podľa najlepších chromozómov každej generácie je tiež možné získať sadu ideálnych testovacích aplikácií (regresnú sadu), pomocou ktorých je možné rýchlo dosiahnuť požadované pokrytie pre daný procesor. Takéto aplikácie sú vhodne pre regresné testovanie procesora po drobných zmenách.

Takisto je z obrázku vidieť, že základné prostredie zostáva nezmenené a GA je integrované len ako istá nadstavba tohto prostredia. Prostredie sa mení hlavne na úrovni triedy test, ktorá je v prípade GA implementovaná ako GA Test. Toto je jadro genetického algoritmu. GA zbiera informácie pomocou pokrytia získaného monitormi jednotlivých agentov verifikačného prostredia. Na základe týchto informácií vytvára nové chromozómy ktoré posiela generátoru. Aplikácia vygenerovaná generátorom sa pomocou komponentu *program loader* nahrá do pamäti, odkiaľ je čítaná procesorom. Z toho vyplýva, že zvyšná časť verifikačného prostredia ostáva bez zásahu, čo umožňuje zachovanie všetkých princípov metodiky UVM.

Pred spustením verifikácie pomocou GA je nutné nastaviť základné parametre a konfiguráciu GA pre každý verifikovaný procesor. Trieda *chromosome* musí byť upravená podľa inštrukčnej sady a požiadaviek pre daný procesor. Tým sa myslí hlavne úprava veľkosti poľa, čiže tela chromozómu podľa počtu inštrukcií a jeho interpretácia pre nastavenie parametrov generátora aplikácií. Je pri tom dôležité meniť štruktúru len do takej miery, aby zmena neovplyvnila operácie mutácie a kríženia a tým aj celý chod GA. Počet chromo-

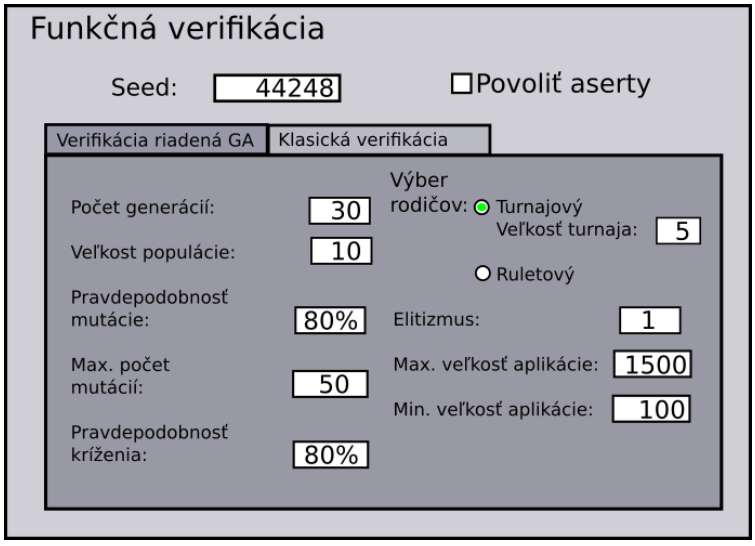


Obr. 4.3: Verifikačné prostredie riadené GA

zómov, ktoré sa dostanú do ďalšej generácie sa nastavuje pomocou parametru *ELITISM*. Použitý GA taktiež umožňuje výber medzi dvoma základnými algoritmami výberu rodičov ďalšej generácie a to pomocou parametru *SELECTION*. Je možné vybrať buď turnajovú selekciu alebo výber tzv. *roulette wheel* algoritmom. Pri turnajovom výbere je nutné nastaviť aj parameter *TOURNAMENT_SIZE*, ktorý vyjadruje počet chromozómov vybratých do "turnaja". Ďalej je nutné nastaviť pravdepodobnosti jednotlivých procesov. Parameter *CROSSOVER_PROB* udáva pravdepodobnosť kríženia a parameter *MUTATION_MAX* a *MUTATION_PROB* udávajú maximálny počet mutácií a ich pravdepodobnosť. Takisto je potrebné nastaviť *POPULATION_SIZE* čiže veľkosť populácie a *GENERATIONS* čiže počet vyhodnotených generácií. Tieto parametre sú veľmi dôležité a na ich vhodnom nastavení závisí efektivita výslednej verifikácie. Tieto hodnoty je možné odhadnúť podľa veľkosti prehľadávaného stavového priestoru, ale často sa najideálnejšie hodnoty zisťujú až experimentálnym skúšaním.

4.3 Návrh automatizácie verifikačného prostredia riadeného GA

Hlavná myšlienka tejto práce je zautomatizovať proces konfigurácie genetického algoritmu a jeho zapojenia do verifikačného prostredia pre procesory ASIP od spoločnosti Codasip. Vzhľadom na to, že použitý GA je navrhnutý pre verifikáciu rôznych druhov integrovaných obvodov, je jedným z cieľov tejto práce aj optimalizácia tohto genetického algoritmu práve pre procesory typu ASIP. Tým sa myslí hlavne štruktúra chromozómu, jej interpretácia a komunikácia s generátorom náhodných aplikácií. Nastavovanie požadovaných parametrov GA bude prebiehať pomocou grafického užívateľského rozhrania (GUI) v prostredí Codasip Studio, pričom východzie nastavenia budú prebrané z práce [16]. Na obrázku 4.4 je znázornený návrh takéhoto GUI.



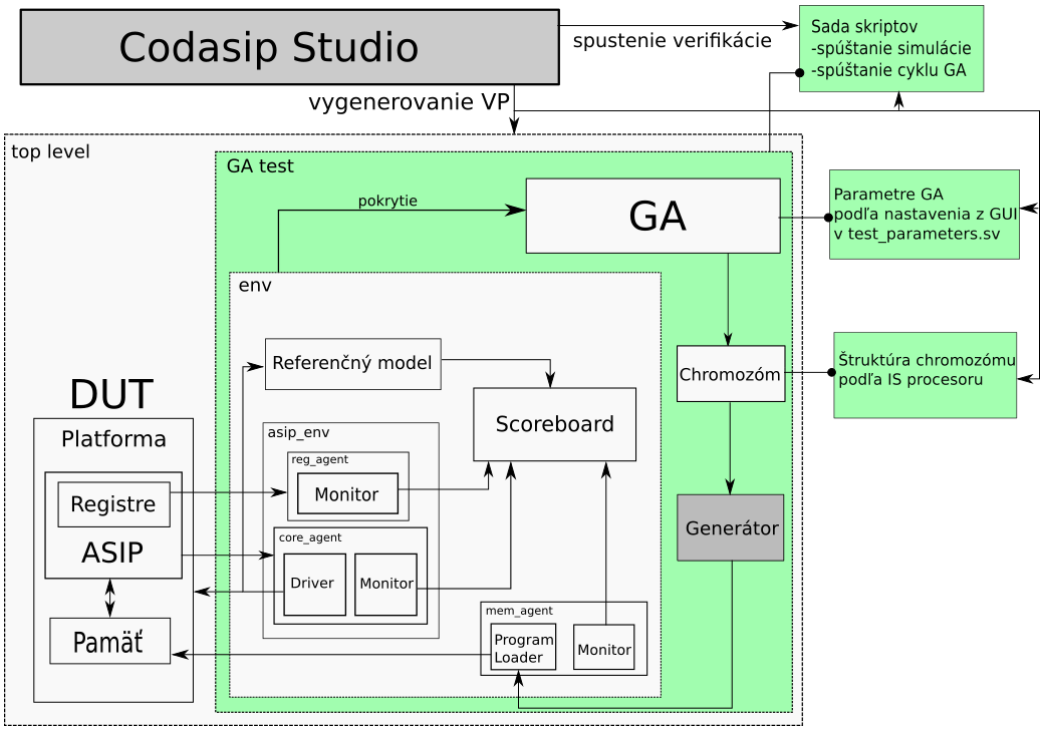
Obr. 4.4: Návrh rozhrania pre verifikáciu riadenú GA

Užívateľ bude mať možnosť si zvoliť medzi klasickým verifikačným prostredím a prostredím riadeným pomocou GA. Po vybratí možnosti s GA, mu bude sprístupnená možnosť konfigurácie klasických parametrov, ako aj tých špecifických pre genetický algoritmus. Tieto

parametre budú prenesené do vygenerovaného prostredia ako parametre GA popísané v kapitole 4.2. Jednotlivé parametre bude možné nastaviť manuálne pomocou GUI ako je vidieť na obrázku 4.4. V oboch prípadoch sa tieto parametre objavia spolu s parametrami verifikácie v súbore *test_parameters.sv*. V tomto súbore bude možné ich ďalej upravovať podľa dosiahnutých výsledkov bez nutnosti veľkého zásahu do zdrojových kódov alebo nutnosti pregenerovať verifikačné prostredie. Ďalej je možné v GUI zvoliť metriky pokrytia, ktoré sa budú pri verifikácií merať. Možnosti pre GA sú funkčné pokrytie, pokrytie kódu a tiež pokrytie na úrovni inštrukcií.

Dôležitou časťou automatizácie verifikačného prostredia pomocou GA je automatická úprava štruktúry chromozómu. Keďže je štruktúra chromozómu silne závislá od inštrukčnej sady procesora, je nutné si najprv zistiť potrebné informácie z definície procesoru. Informácie o inštrukčnej sade je možné získať zo súboru generovaného CS podľa definície procesoru podobne ako to robí generátor náhodných aplikácií. Takisto bude upravené jadro GA podľa parametrov a konfigurácie nastavenej pomocou GUI.

Aby bolo možné spustiť verifikáciu riadenú GA je nutné vytvoriť všetky potrebné prepojenia medzi nadstavbou prostredia obsahujúcou genetický algoritmus a štandardným verifikačným prostredím. Tým sa myslí hlavne spracovanie a odosielanie dosiahnutého pokrytia genetickému algoritmu a tiež úprava skriptov spúšťajúcich verifikáciu a následnú simuláciu. Tieto skripty sa budú starať aj o spustenie procesu genetického algoritmu a o spúšťanie verifikácie pre každý chromozóm podľa pokynu GA. Ďalšou úlohou týchto skriptov je zabezpečiť konzistenciu verifikačného prostredia a teda najmä správu konfiguračných a pomocných súborov tak, aby užívateľ pri novom spustení behu genetickým algoritmom riadenej verifikácie nemusel vykonávať žiadne dodatočné úpravy. Celý proces automatizácie je zobrazený na obr. 4.5.



Obr. 4.5: Automatizácia verifikačného prostredia

Zelenou vyznačené oblasti predstavujú tie komponenty v ktorom sa verifikačné prostre-

die riadené GA líši od štandardného prostredia vygenerovaného Codasip Studiom. Tieto časti je teda nutné zahrnúť do procesu generovania verifikačného prostredia pomocou parametrov nastavených v CS. Výsledkom tohto procesu je teda plne automatizované generovanie verifikačného prostredia spolu so zapojeným a nakonfigurovaným genetickým algoritmom, slúžiacim na optimalizáciu verifikačného behu pomocou spätnej väzby dosiahnutého pokrytia pre generovanie nových vstupov. Počas verifikácie sa bude dosiahnuté pokrytie ďalej uchovávať a na konci verifikácie bude spracované a pripravené na analýzu verifikačným inžinierom.

Kapitola 5

Realizácia riešenia

Finálne riešenie tejto práce pozostáva z verifikačného prostredia a doňho zapojeného genetického algoritmu, ktorý riadi beh verifikácie. Toto prostredie je pripravené na integráciu do vývojového prostredia Codasip Studio. V tejto kapitole sú popísané ručné úpravy, ktoré bolo nutné previesť aby bolo toto verifikačné prostredie univerzálne a tým pádom mohlo byť automaticky generované pre rôzne ASIP procesory navrhnuté v CS.

5.1 Automatizácia verifikácie riadenej pokrytím

Základom finálneho verifikačného prostredia je prostredie vygenerované pomocou CS. Na základe rozšírení navrhnutých v tejto práci je do tohto prostredia ďalej zapojený GA, ktorý verifikáciu riadi a generátor náhodných aplikácií, ktorý na základe spätnej väzby od GA generuje ďalšie aplikácie pre verifikovaný ASIP procesor. Hlavnými zmenami oproti pôvodnému verifikačnému prostrediu je integrácia riadiaceho algoritmu GA do UVM triedy *test* v súbore *test.svh*, zaistenie správneho nahrávania vygenerovaných aplikácii do verifikovaného procesoru a jeho referenčného modelu a tiež úprava skriptov riadiacich beh verifikácie v simulátore. Do prostredia bola taktiež integrovaná knižnica implementujúca triedy, v ktorých je definovaný chromozóm a operácie s ním, konkrétne v súbore *codix_risc_chromosome.svh*. Ďalej bola implementovaná trieda zabezpečujúca komunikáciu a riadenie generátora pseudonáhodných aplikácií nazvaná *Generator* v súbore *generator.svh*. Aby mal GA dostatočné informácie o kvalite jednotlivých chromozómov, čiže aj aplikácií vygenerovaných na základe nich, bolo potrebné rozšíriť meranie pokrytia v súbore *coverage.svh*. Pokrytie teda zahŕňa kompletné inštrukčné pokrytie, pokrytie sekvencií jednotlivých inštrukcií v rôznych typoch procesorov a pokrytie niektorých základných signálov procesoru. Kvôli variabilite možných inštrukcií je niekedy vhodné, na základe výsledkov GA, manuálne upraviť merané pokrytie, čiže vylúčiť z neho nepovolené inštrukcie alebo sekvencie inštrukcií, či nedosiahnuteľné hodnoty niektorých signálov. Toto zabezpečí optimálny beh verifikácie. Názorné príklady a výsledky, budú rozobraté v nasledujúcej kapitole.

GA rozšírenia pre verifikačné prostredie boli navrhnuté tak, aby mohlo byť použité univerzálne pre rôzne procesory navrhnuté pomocou CS, bez veľkých zásahov potrebných na ich zapojenie. Jedinými zmenami potrebnými pre chod genetickým algoritmom riadenej verifikácie sú nastavenie veľkosti chromozómu a nahratím potrebných nástrojov (asembler a linker) špecifických pre konkrétny procesor. V budúcnosti budú tieto zmeny riešené automaticky generátorom verifikačného prostredia v CS.

Aby bolo možné do verifikácie riadenej pokrytím pomocou GA zapojiť rozličné proce-

sory je potrebné nastaviť veľkosť chromozómu tak, aby zodpovedal inštrukčnej sade daného procesoru. Toto je zabezpečené pomocou skriptu, ktorý dokáže z modelu daného procesora navrhnutého v CS vyčítať počet inštrukcií a nastaviť potrebné parametre v súbore *test_parametres.svh*. Príklad možných nastavení v tomto súbore je vidieť na obrázku 5.1.

```
// GENETIC ALGORITHM PARAMETERS
// ASIP specifics
parameter INSTRUCTION_NUMBER = 39;
parameter MIN_PROGRAM_SIZE = 100;
parameter MAX_PROGRAM_SIZE = 1500;

// File with parameters and chromosomes
parameter CHROMOSOMES_FILE = "chromosomes.txt";
parameter BEST_CHROMOSOMES_FILE = "best_chromosomes.txt";

// File with fitness values
parameter FITNESS_FILE = "fitness.txt";
parameter PROCESSOR_NAME = "codasip_urisc";
// Number of generations
parameter GENERATIONS      = 30;
// Size of population,/cle»
parameter POPULATION_SIZE  = 10;
// Elitism
parameter ELITISM          = 1;
// Selection
parameter SELECTION        = 1; // 0 == proportionate, 1 == tournament
parameter TOURNAMENT_POOL_SIZE = 5;

// Crossover probability
parameter CROSSOVER_PROB   = 80;
// Mutation probability
parameter MUTATION_PROB    = 70;

// Number of maximal mutations per individuum
parameter MAX_MUTATIONS    = 40;

// File for save/load population
parameter string POPULATION_FILENAME = "pop";
// Load or create new population on evolution start
parameter LOAD_POPULATION  = 0;
parameter ESTIMATED_TIME   = 800;

// test parameters
// how many transaction to generate or limit for detection of halt instruction
parameter TRANSACTION_COUNT = 1000000;
// limits the number of miscompares reported in the output file
parameter MAX_ERROR_COUNT = 100;
// initial seed for the PRNG
parameter SEED = 1445545763;
// timeout detection
parameter string AGENT_TIME_OUT = "agent_time_out";
// finish checking limit (limit for detection of finish signal for sequential units)
parameter ITERATION_COUNT_LIMIT = 1000000;
```

Obr. 5.1: Príklad možných nastavení v súbore *test_parametres.svh*

Asembler a linker sú nástroje potrebné na kompiláciu vygenerovanej aplikácie. Tieto nástroje je možné automaticky vygenerovať a vyexportovať pre konkrétny model procesoru v prostredí Codasip Studio. Po nahratí do verifikačného prostredia sú zapojené do automatizovaného behu verifikácie optimalizovanej pomocou GA.

5.2 Generovanie pseudo-náhodných aplikácií

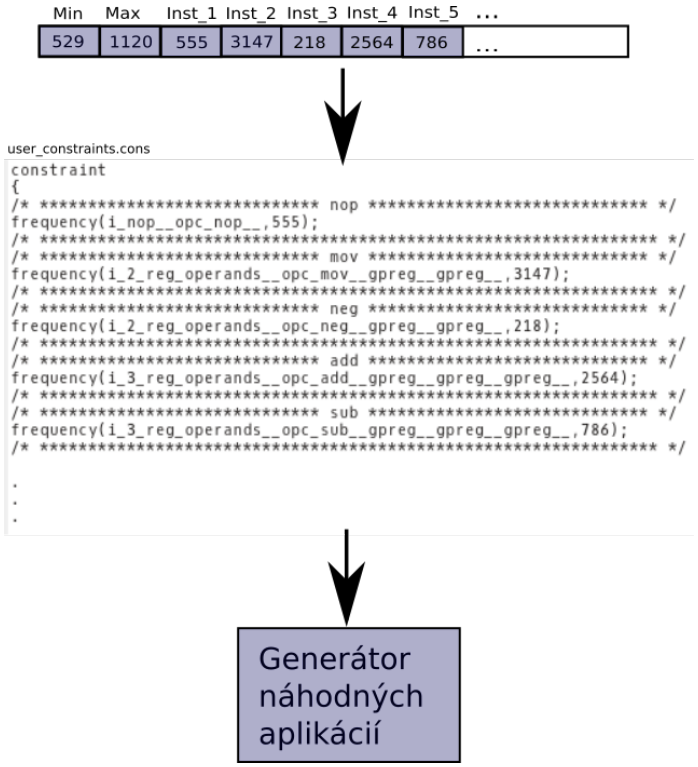
Dôležitou súčasťou verifikačného prostredia riadeného pomocou GA je generovanie náhodných aplikácií a možnosť toto generovanie ovplyvňovať za behu na základe dosahovaných výsledkov. V riešení tohto problému bola použitá najnovšia verzia generátoru aplikácií, ktorá je súčasťou vývojového prostredia Codasip Studio.

Generátor aplikácií generuje náhodné aplikácie pre špecifický procesor popísaný v jazyku CodAL v prostredí CS. Na základe zdrojového kódu, ktorý opisuje inštrukčnú sadu daného procesoru, je CS schopné vygenerovať sémantický popis inštrukcií a základných obmedzení ich použitia ako je latencia alebo podmienky skokových inštrukcií. Tieto obmedzenia inštrukčnej sady vygenerované CS sú obsiahnuté v súbore *basic_constraints.cons* a sú potom vstupom pre generátor, ktorý dokáže na základe nich vygenerovať validné aplikácie formou zdrojového kódu v asembleri daného procesoru. Pre komplexnejšie inštrukcie, ktorých použitie si vyžaduje splnenie zložitejších podmienok, môže byť nutné dopísať niektoré

užívateľské obmedzenia pre zaistenie validity vygenerovaných programov podľa špecifikácie procesora. Užívateľské obmedzenia sa nachádzajú v súbore *user_constraints.cons* a sú štandardne súčasťou modelu procesoru v prostredí CS. Pri verifikačnom prostredí riadenom pomocou GA, sú tieto súbory integrované do verifikačného prostredia, pretože aj samotný generátor aplikácií je v tomto prípade priamou súčasťou prostredia.

5.2.1 Riadenie generátora z verifikačného prostredia

Celú komunikáciu s generátorom aplikácií zabezpečuje trieda *Generator* implementovaná v súbore *generator.svh*. Do verifikačného prostredia riešeného v tejto práci je generátor zapojený vo forme dynamickej knižnice jazyka C. Táto knižnica obsahuje funkcie potrebné na riadenie generátora a takisto na jeho inicializáciu a alokáciu potrebných zdrojov a je súčasťou CS. S verifikačným prostredím komunikuje pomocou rozhrania DPI (Direct Programming Interface), ktorá umožňuje volanie funkcií jazyka C/C++ z prostredia jazyka SystemVerilog a opačne.



Obr. 5.2: Proces nastavenia obmedzení pre generátor

Ovplyvňovanie procesu generovania aplikácií prebieha ako prepisovanie užívateľských obmedzení priamo počas verifikácie na základe dát v chromozóme, a to pomocou skriptu jazyka Python *parser.py*. Tento skript je potom volaný funkciou jazyka C pomocou knižnice *Python.h* poskytovanej v vývojárskych nástrojoch jazyka Python, pretože jazyk SystemVerilog nepodporuje priame volanie funkcií napísaných v jazyku Python. Funkcia jazyka C je potom následne volaná pomocou rozhrania DPI. Ako vstup pre skript upravujúci užívateľské obmedzenia slúži reťazec predstavujúci celý chromozóm na základe ktorého sa má program vygenerovať. Chromozóm obsahuje váhy s akými sa majú jednotlivé inštrukcie procesoru vygenerovať zakódované v binárnej forme. Skript tieto váhy spracuje a prenesie

do súboru *user_constraints.cons* s ktorým už pracuje samotný generátor. Celý proces je graficky znázornený na obrázku 5.2.

Výsledným produktom generátora je teda súbor obsahujúci aplikáciu pre daný procesor. Aby bolo možné túto aplikáciu nahrať do verifikovaného procesora, je potrebné aplikáciu spracovať nástrojmi asembler a linker, ktoré sú špecifické pre daný procesor.

5.3 Simulácia a výsledky

O kompiláciu zdrojových súborov a beh verifikácie sa stará sada Tcl skriptov pre simulátor QuestaSim vygenerovaných CS. Tieto súbory boli upravené tak aby zahrňovali aj dodatočnú implementáciu a zabezpečili správny chod verifikácie a genetického algoritmu. Výsledkom behu verifikácie v simulátore je správa o výsledkoch a dosiahnutom pokrytí v súbore *final_report.txt*. Pri výskyte nezhody medzi RTL a referenčným modelom procesora je takáto nezhoda zaznamenaná v tomto súbore spolu s aplikáciou pri ktorej chyba vznikla. Táto aplikácia, ako aj súbor s jej zdrojovým kódom, je potom dostupná spolu s ostatnými v priečinku xexes. Do výslednej správy sa dostanú aj prípadné chyby pri kompilácií alebo behu simulácie. V súbore *best_chromosomes.txt* je možné získať informáciu o aplikáciách, ktoré spolu získali najväčšie pokrytie. Tieto aplikácie môžu byť použité na tvorbu sady testov pre regresné testovanie procesora, dosahujúcich vysoké pokrytie bez nutnosti spúšťať celý proces verifikácie riadenej GA. Súčasťou prostredia je aj skript zabezpečujúci vyčistenie verifikačného prostredia pre jeho opätovný beh.

Kapitola 6

Dosiahnuté výsledky

Oproti pôvodnému riešeniu z [16] je verifikačné prostredie univerzálne a je možné doňho zapojiť rôzne procesory navrhnuté pomocou CS s minimálnym počtom potrebných úprav. Účinnosť verifikačného prostredia bola experimentálne overená a zmeraná na dvoch procesoroch typu ASIP vyvinutých spoločnosťou Codasip.

Prvým testovaným procesorom bol procesor Codasip uRISC. Tento procesor je vyvinutý najmä pre výukové účely, ale obsahuje kompletnú minimálnu inštrukčnú sadu potrebnú pre automatické zostavenie kompilátora jazyka C. Ďalším testovaným procesorom je Codix Cobalt, ktorý obsahuje kompletnú RISCovú inštrukčnú sadu a je určený ako procesor pre všeobecné použitie s vysokým výkonom.

Experimenty z pôvodného riešenia GA boli zopakované pre ďalšie procesory a pomocou získaných výsledkov boli navrhnuté a prevedené aj nové experimenty s GA. Nová verzia generátora náhodných aplikácií a verifikačného prostredia značne ovplyvnila priebeh verifikácie so zapojeným GA, ale napriek tomu ukázala viditeľne lepšie výsledky oproti aplikáciám vygenerovaným generátorom bez spätnej väzby od GA. Nová verzia generátora umožňuje taktiež konfiguráciu generovania náhodných aplikácií, preto je možné vykonať experimenty s rôznymi nastaveniami a nájsť tak čo najideálnejšie riešenie.

6.1 Pokrytie

Po vykonaní prvých experimentov bolo nutné pre oba testované procesory ručne upraviť ciele dosiahnutého pokrytia vo verifikačnom prostredí. Na obrázku 6.1 je vidieť, že niektoré sekvencie inštrukcií pre procesor uRISC nikdy nenastali ani po dlho-trvajúcom testovaní. Po dôkladnej analýze sa zistilo, že dôvodom je funkcionalita generátora náhodných aplikácií, ktorá zaisťuje, aby boli pred vykonaním všetkých inštrukcií splnené všetky podmienky pre jej vykonanie vyplývajúce zo špecifikácie. Kvôli tomuto je potrebné pred vygenerovaním inštrukcie vykonať inicializáciu registrov s ktorými bude daná inštrukcia pracovať. Týka sa to najmä všetkých skokových inštrukcií a inštrukcií pracujúcich s pamäťou, aby nedošlo k prepísaniu programovej pamäti alebo skoku mimo rozsah aplikácie. Po analýze dosiahnutého pokrytia v prvých experimentoch boli z cieleného pokrytia pre oba procesory odstránené nedosiahnuteľné sekvencie inštrukcií. Na obrázku 6.2 je zobrazený úsek súboru *coverage.svh* a nastavenie obmedzení pre požadované pokrytie v jazyku SystemVerilog. Týmto bolo zaistené, aby dosiahnuté pokrytie z ďalších experimentov reálne odrážalo efektivitu GA.

Obr. 6.1: Úsek výsledného pokrytia pre inštrukciu JUMP

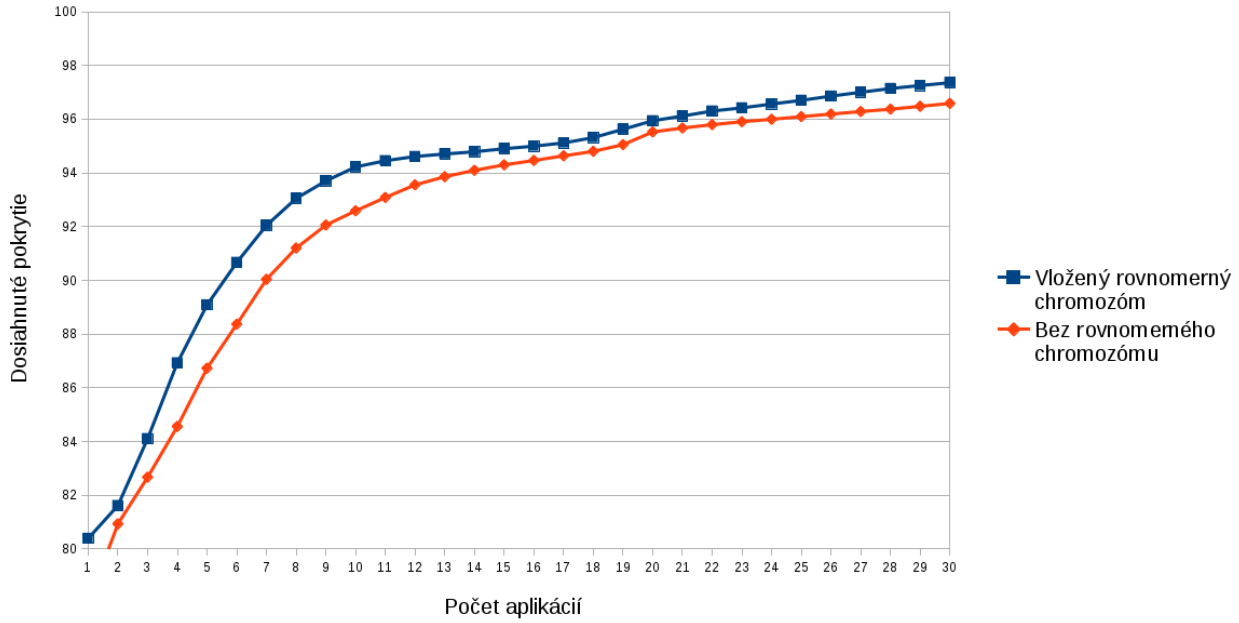
```
// combinations of instructions
inst_after_inst: coverpoint m_transaction_h.m_instruction {
// unknown is illegal
illegal_bins unknown_trans1 = ([0:$] => {UNKNOWN});
illegal_bins unknown_trans2 = ({UNKNOWN} => [0:$]);

ignore_bins load = ([0:$] => [LOAD___dest____base_____imm16__:LOADUH___dest____base_____imm16__]);
ignore_bins store = ([0:$] => [STORE___source____base_____imm16__:STOREH___source____base_____imm16__]);
ignore_bins jump_source = ([0:$] => {JUMP___source__});
ignore_bins call_source = ([0:$] => {CALL___source__});
ignore_bins halt = ({HALT_} => [0:$]);

//all other options
bins inst_after_inst[] = ([0:$] => [0:$]);
option.at_least = 10;
}
```

Obr. 6.2: Obmedzenie cieľov pokrytia v súbore *coverage.svh*

6.2 Experimenty

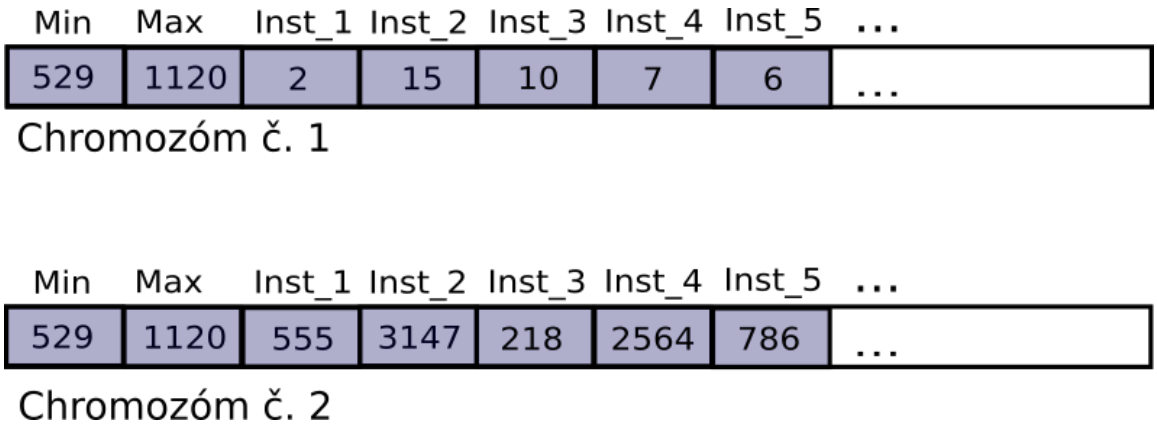
Experimenty pre oba testované procesory sú rozdelené do viacerých kategórií. V prvej sa zameriavame na generovanie náhodných aplikácií bez použitia genetického algoritmu. V ďalších experimentoch sú využité nastavenia generátora a to hlavne rozptyl medzi váhami s akými sa jednotlivé inštrukcie generujú. Ukázalo sa, **že vysoké rozdiely medzi váhami mali negatívny vplyv na celý priebeh verifikácie** pomocou GA a tento prístup zvýhodňoval verifikáciu bez zapojeného GA a rovnomerne nastavenými váhami. Riešením tohto problému bolo zahrnúť do počiatočnej populácie chromozóm s vyváženými hodnotami váh tak, aby aplikácia vygenerovaná na základe tohto chromozómu mala rovnomerné rozloženie všetkých inštrukcií. Graf 6.3 znázorňuje výsledok tohto experimentu s vloženým "rovnomerným"chromozómom a bez neho pre procesor uRISC. Tento chromozóm mal obvykle vo svojej populácii najväčšiu fitness, čiže dosiahnuté pokrytie, a umožnil tak GA pracovať s chromozómmi s oveľa vyššou variabilitou. Toto viedlo k zvýšeniu celkového dosiahnutého pokrytia hlavne v koncových fázach verifikácie.

Nastavenie generátora s malými rozptylami medzi váhami (č. 1), oproti pôvodnému chromozómu (č. 2), ako je vidieť na obrázku 6.4, sa ukázalo výhodné najmä v pokrývaní vyšších počtoch možných sekvencií, teda hlavne u procesoru Cobalt. Experimenty tiež ukázali, že použitie chromozómu s vyváženými váhami má pri menších rozptyloch výrazne



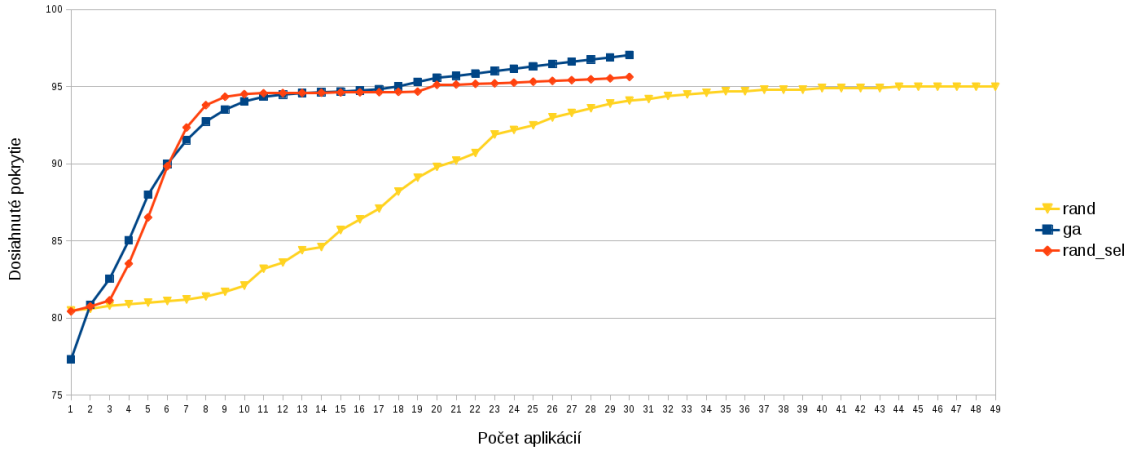
Obr. 6.3: Porovnanie účinnosti GA s nasadeným rovnomerným chromozómom a bez neho

menší vplyv. Takéto nastavenie dáva vyššiu možnosť vygenerovaniu inštrukcií s menšou váhou a preto dosahuje lepšie výsledky pri pokrývaní veľkého množstva rôznych sekvencií.



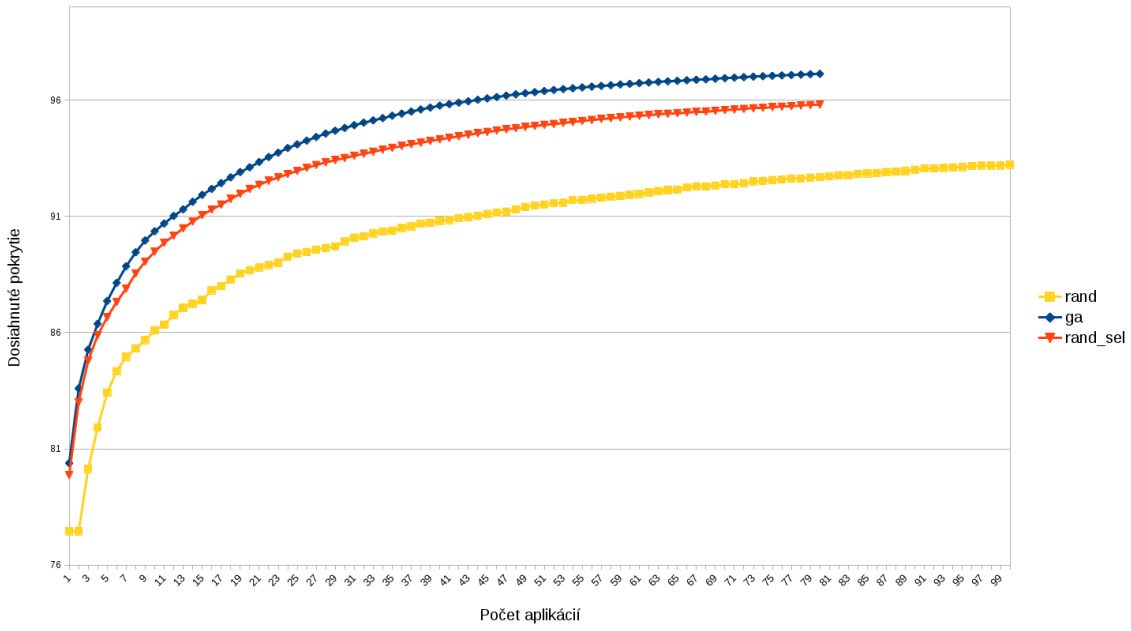
Obr. 6.4: Chromozóm s malými rozptylmi medzi váhami a pôvodný chromozóm

Na obrázku 6.5 je zobrazený graf porovnávajúci priebeh verifikácie riadenej GA (označený v grafe ako "ga") oproti verifikácií len s generátorom náhodných aplikácií pre procesor uRISC. Verifikácia bez zapojeného GA je rozdelená na dva typy. Prvým je vygenerovanie skupiny aplikácií a následný výber najlepšej z nich ako základ pre nasledujúcu skupinu. Tento prístup je podobný princípu populáciám a generáciám GA, ale nové aplikácie nevznikajú na základe spätnej väzby od GA ale čisto náhodne (v grafe označené ako "rand_sel"). Tento prístup sa však bežne nepoužíva a v týchto experimentoch slúži len na zdôraznenie účinnosti GA. Druhým typom je klasická verifikácia pomocou generátora náhodných aplikácií, t.j. vygenerovanie sady aplikácií a ich následná verifikácia a zmeranie dosiahnutého pokrytia (označený v grafe ako "rand").



Obr. 6.5: Porovnanie účinnosti verifikácie riadenej GA pre procesor uRISC

Na obrázku 6.6 je vidieť graf znázorňujúci to isté porovnanie pre procesor Cobalt. Z grafov je vidieť, že napriek tomu, že generátor bez zapojeného GA a výberom najlepšieho zo skupiny náhodných aplikácií má lepšie výsledky na začiatku verifikácie, genetický algoritmus dosahuje vyššie pokrytie celkovo. Ukazuje sa, že GA pomáha dosiahnuť najmä okrajové sekvencie a vzácnejšie kombinácie inštrukcií, ktoré už generátor aplikácií bez spätnej väzby od GA nedokáže pokryť. Napriek tomu, že percentuálne zvýšenie pokrytia nie je radikálne vysoké, práve takéto okrajové prípady sú v praxi najväčším zdrojom neodhalených chýb. Oproti klasickému prístupu generovania náhodných aplikácií dosahuje navyše verifikácia riadená pokrytím pomocou genetického algoritmu výrazne lepšie výsledky. Nespornou výhodou je tiež možnosť získania regresnej testovacej sady, ktorá je výhodná najmä po dodatočných menších úpravách v procesore.



Obr. 6.6: Porovnanie účinnosti verifikácie riadenej GA pre procesor Cobalt

Experimenty boli prevádzané s parametrami GA nastavenými podľa práce [16]. Všetky merania boli vykonané pomocou simulátora QuestaSim od spoločnosti Mentor Graphics.

Kapitola 7

Záver

Hlavným cieľom práce bola automatizácia verifikácie riadenej pokrytím pre procesory typu ASIP v prostredí Codasip Studio. Na optimalizáciu verifikácie na základe dosiahnutého pokrytia bol použitý genetický algoritmus navrhnutý v [16]. Hlavnými bodmi realizácie riešenia bola úprava komponentov verifikačného prostredia navrhnutého podľa metodiky UVM tak, aby bolo možné doňho zapojiť genetický algoritmus, ktorý bude verifikáciu riadiť. Ďalej bolo nutné zaistiť prepojenie a správnu komunikáciu genetického algoritmu s novou verziou generátora náhodných aplikácií od spoločnosti Codasip. Požiadavkou na finálne riešenie tiež bolo, aby výsledné verifikačné prostredie so zapojeným genetickým algoritmom bolo použiteľné univerzálne bez nutnosti väčších zmien pre rôzne ASIP procesory, a tým pripravené na integráciu do vývojového prostredia Codasip Studio. Tento cieľ bol splnený a výsledky boli experimentálne zmerané a overené pre procesory Codasip uRISC a Codix Cobalt poskytnuté spoločnosťou Codasip. Zmerané výsledky ukázali účinnosť verifikácie riadenej pokrytím oproti tradičnej funkčnej verifikácií. Výhodou je aj možnosť získania sady testovacích aplikácií pre daný procesor, ktoré pri opätovnom použití rýchlo dosiahnu požadované pokrytie a môžu byť použité na regresné testovanie procesora po dodatočných úpravách.

Pripravené verifikačné prostredie spolu so zapojeným genetickým algoritmom ponúka vysoké možnosti konfigurácie celého procesu verifikácie a preto je možné ďalším experimentálnym meraním pre rôzne nastavenia, získať ešte lepšie výsledky. Výsledné riešenie je tiež pripravené na integráciu do Codasip Studia a tým pádom na zautomatizovanie celého procesu generovania verifikačného prostredia so zapojeným genetickým algoritmom pre všetky procesory typu ASIP navrhnuté v CS. Ďalším možným rozšírením tejto práce môže byť aj automatický odhad ideálnej konfigurácie GA podľa inštrukčnej sady procesoru alebo optimalizácia rýchlosti behu celého procesu.

Literatúra

[1] ARAUJO, P.: *Development of a reconfigurable multi-protocol verification environment using UVM methodology*. Diplomová práce, Universidade do Porto, 2014.

[2] BABU, S.; BARTLEY, M.: *A Comparison of OVM and UVM*. 2013, [Online; navštíveno 12.5.2016].
URL http://www.soccentral.com/results.asp?CatID=488&EntryID=40650

[3] DARWIN, C.: *On the Origin of Species b*. Murray, 1861.
URL https://books.google.cz/books?id=SmFEAAAAcAAJ

[4] EIBEN, A.; SMITH, J.: *Introduction to Evolutionary Computing*. Natural Computing Series, Springer Berlin Heidelberg, 2013, ISBN 978-3-6-6205094-1.
URL https://books.google.cz/books?id=ssmqCAAAQBAJ

[5] FOGEL, L.; OWENS, A.; WALSH, M.: *Artificial Intelligence Through Simulated Evolution*. John Wiley & Sons, 1966.
URL https://books.google.cz/books?id=QMLaAAAAMAAJ

[6] FOSTER, H. D.: Trends in functional verification: A 2014 industry study. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015, ISSN 0738-100X.

[7] HEIGHT, H.: *A Practical Guide to Adopting the Universal Verification Methodology (UVM) Second Edition*. Lulu.com, 2010, ISBN 9781300535935.
URL https://books.google.sk/books?id=EqwOBAAAQBAJ

[8] HOLLAND, J.: *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press, 1975, ISBN 978-0-4-7208460-9.
URL https://books.google.cz/books?id=JE5RAAAAMAAJ

[9] KOZA, J.: *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. A Bradford book, Bradford, 1992, ISBN 978-0-2-6211170-6.
URL https://books.google.de/books?id=Bhtxo60BV0EC

[10] MITCHELL, M.: *An Introduction to Genetic Algorithms*. A Bradford book, Bradford Books, 1998, ISBN 978-0-2-6263185-3.
URL https://books.google.cz/books?id=0eznlz0TF-IC

[11] RACHENBERG, I.: *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Problemata, 15, Frommann-Holzboog, 1973, ISBN 978-3-7-7280373-4.
URL https://books.google.de/books?id=-WAQAQAAMAAJ

[12] SCHWEFEL, H.: *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie: Mit einer vergleichenden Einführung in die Hill-Climbing- und Zufallsstrategie*. Interdisciplinary Systems Research, Birkhäuser Basel, 1976, ISBN 978-3-7-6430876-6.
URL https://books.google.cz/books?id=AP5ZAAAACAAJ

[13] SELIGMAN, E.; SCHUBERT, T.; KUMAR, M. V. A. K.: *Formal Verification*. Elsevier Inc., 2015, ISBN 978-0-2-800727-3.

[14] SPEAR, C.; TUMBUSH, G.: *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*. Springer US, 2012, ISBN 978-1-4-6140715-7.
URL https://books.google.sk/books?id=QaWOYTOXy0EC

[15] WILE, B.; GOSS, J.; ROESNER, W.: *Comprehensive Functional Verification: The Complete Industry Cycle*. Systems on Silicon, Elsevier Science, 2005, ISBN 978-0-0-8047664-3.
URL https://books.google.sk/books?id=btl_OX3kJ7MC

[16] ZACHARIÁŠOVÁ, M.; KOTÁSEK, Z.: Automation and Optimization of Coverage-driven Verification. In *Proceedings of the 18th Euromicro Conference on Digital Systems Design*, IEEE Computer Society, 2015, ISBN 978-1-4673-8035-5.
URL http://www.fit.vutbr.cz/research/view_pub.php.cs?id=10951